Analysis of Bubble Sort

We began with an example featuring bubble sort. During the discussion I had to correct the code I had written on the board. And now I see that one more correction was necessary: The initial value on the index in the inner loop should be 2, not 1. Here is that corrected code:

1 Bubble-sort(A)
2   for i = 1 to n
3     for j = 2 to n-i+1
5   end
6 end Bubble-sort

Loop invariant: At the start of each iteration of the for-loop of lines 2-6, the subarray $A[n - i + 2..n]$ contains the largest $i - 1$ values in the array and they’re sorted.

I explained how each of the properties for a loop invariant hold for this one.

Initialization property: Show that the invariant holds before entering the loop the first time. At that point, the variable $i$ has the value 1, so the invariant is a statement about the subarray $A[1..n]$. With those indices the subarray is empty so the rest of the statement is vacuously true.

Maintenance: Show that if the invariant holds when the value of $i$ is some value $k$ just before the loop test that it holds after one trip through the loop, that is, that it holds when $i$ is $k + 1$ just before the loop test. This means that we assume that before executing the loop body with $i = k$, the largest $k - 1$ values originally in the array are now in order in the last $k - 1$ entries of the array, that is, in $A[n - k + 2..n]$. I argued that, as the inner loop iterates, it encounters the largest value in the subarray $A[1..n-k+1]$ and proceeds to swap it down to position $n - k + 1$. And in so doing leaving the largest $k$ values originally in the array sorted and in the subarray $A[n-k+1..n]$.

Termination: Show that if the invariant holds after the loop has terminated then the algorithm has done what we claim it is supposed to do, which, in this case, that it has sorted the array. When the loop terminates, the value of $i$ is $n + 1$. Plugging this in for $i$ in the invariant statement, we get the statement that the subarray $A[1..n]$ contains the largest $n$ values in the array and they’re sorted.
I then analyzed the running time by looking at how often the comparison at line 4 is executed. I did this by translating loop indices and limits into summations (here corrected with the \(j\) starting at 2):

\[
\sum_{i=1}^{n} \sum_{j=2}^{n-i+1} 1 = \sum_{i=1}^{n} n - i \\
= \sum_{i=1}^{n} n - \sum_{i=1}^{n} i \\
= n^2 - \frac{n(n+1)}{2} \\
= n^2 - \frac{1}{2}n^2 - \frac{1}{2}n \\
= \frac{1}{2}n^2 - \frac{1}{2}n = \Theta(n^2)
\]

Asymptotic notation (chapter 3)

I explained that the purpose of asymptotic notation (i.e., big-Oh, big-Omega, big-Theta) is to allow comparison of the running times of algorithms in a way that captures the practical efficiency of each.

Using the definitions in the text, I defined the notations \(O\), \(\Omega\), and \(\Theta\), emphasizing that each represents a set of functions. For example, \(O(n^2)\) is the set of all functions that asymptotically grow now faster than the function \(n^2\).

For example, to show that \(\frac{1}{2}n^2 + \frac{1}{2}n\) is \(O(n^2)\), we need, as the definition of \(O\) specifies, a constant \(c\) and an integer \(n_0\) such that

\[
\frac{1}{2}n^2 + \frac{1}{2}n \leq cn^2, \forall n \geq n_0
\]

As the following shows, letting \(c = 1\) and \(n_0 = 0\), does the trick:

\[
\frac{1}{2}n^2 + \frac{1}{2}n \leq \frac{1}{2}n^2 + \frac{1}{2}n^2 = n^2, \forall n \geq 0
\]

To show that that \(\frac{1}{2}n^2 + \frac{1}{2}n\) is \(\Omega(n^2)\), we need, as the definition of \(\Omega\) specifies, a constant \(c\) and an integer \(n_0\) such that

\[
\frac{1}{2}n^2 + \frac{1}{2}n \geq cn^2, \forall n \geq n_0
\]

As the following shows, letting \(c = \frac{1}{2}\) and \(n_0 = 0\) works:
Chapter 4: Divide and conquer algorithms

I briefly explained where the term “divide and conquer” comes from but most people had heard that previously. The canonical example of such algorithms is merge sort. I forbore explaining it as, again, many had already seen this.

Instead, I explained the problem mentioned in section 4.1 of the text: The maximum subarray sum problem. A detailed explanation is in the book so I summarized it: You have an array of numbers $A[1..n]$. You want to find the indices of a subarray (i.e., two values $i \leq j$) such that the sum of the values in $A[i..j]$ is the maximum over all subarrays. Because each subarray is defined by its starting and ending indices, the number of subarrays is equal to the number of ways I can select two values from $1..n$. So the brute force algorithm would try all possible pairs. That number is $\binom{n}{2} = \frac{n(n+1)}{2} = \frac{1}{2}n^2 + \frac{1}{2}n = \Theta(n^2)$.

Can we do better? The answer is yes, if we use a divide and conquer approach. Wherever the maximum subarray is, it must fall into one of three places. Let $mid = \lfloor n/2 \rfloor$. The max subarray is either entirely contained in $A[1..mid]$, or entirely contained in $A[mid+1..n]$, or it crosses the midpoint, that is, its first index is less than or equal to $mid$ and its second index is greater than $mid$.

The divide and conquer algorithm is explained on pages 70 to 73 of the text. I won’t repeat it here but will instead turn to its analysis.

Look at the algorithm $\text{Find-Maximum-Subarray}$ on page 72. We want to determine how many additions it performs on values from the array $A$. As you can see, the algorithm performs no additions in its pseudocode. But it calls the algorithm $\text{Find-Max-Crossing-Subarray}$ which performs $n$ additions on an array of length $n$ (that is, in terms of the parameters passed to the algorithm, $n = high - low + 1$).

Let $T(n)$ stand for the number of additions performed by the algorithm. Looking again at page 72, we see that line 6 calls $\text{Find-Max-Crossing-Subarray}$ and so $n$ additions are done there. The other two calls are recursive calls to the algorithm itself. We don’t know how many additions they perform but we do have a term for each: It’s $T(n/2)$.
So we can define $T(n) = n + T(n/2) + T(n/2) = 2T(n/2) + n$. Adding the base case $T(1) = 0$ and we now have a recurrence relation:

\[
\begin{align*}
T(1) &= 0 \\
T(n) &= 2T(n/2) + n
\end{align*}
\]

But this doesn’t tell us much about how to find the asymptotic version of this. For that, we need to solve the recurrence and to do that, I used the substitution method. To make things a little easier to compute, I assumed that $n$ is a power of 2, that is, that $n = 2^k$, for some value $k$. Because dividing a power of 2 by 2 never yields an odd number, we can do without the floor and ceiling functions. So:

\[
T(2^k) = 2T(2^{k-1}) + 2^k
\]

\[
= 2[2T(2^{k-2}) + 2^{k-1}] + 2^k
\]

\[
= 2^2T(2^{k-2}) + 2^k + 2^k
\]

\[
= 2^2T(2^{k-2}) + 2 \cdot 2^k
\]

\[
= 2^2[2T(2^{k-3}) + 2^{k-2}] + 2 \cdot 2^k
\]

\[
= 2^3T(2^{k-3}) + 3 \cdot 2^k
\]

*** At this point notice where the value 3 is popping up. Every time we substitute a term of the form $T(2^j)$ we see the exponent $j$ getting smaller. It starts at $k$ and becomes $k-1$, $k-2$, …, $k-i$. As we go along, the formula looks like:

\[
2^iT(2^{k-i}) + i \cdot 2^k
\]

This bottoms out when the exponent is 0 because then we have the base case $T(2^0) = T(1) = 0$. That happens when $i$ becomes $k$. So we end up with

\[
2^kT(2^{k-k}) + k \cdot 2^k
\]

The left half of the sum is 0 (because $T(1) = 0$) so we’re left with $k \cdot 2^k$. Because $n = 2^k$, we can write this in terms of $n$ as $n \lg n$.

The problem is that the place above that I marked with *** is not a mathematical step. I just happened to notice a pattern that I then assumed carried all the way through. Although I get a closed form for the recurrence, my technique was not completely mathematical. So, as a last step, I used a proof by induction to show that my closed form really is correct.

**Theorem.** $T(n) = n \lg n$

**Base case:** $k = 0$. Then $n = 1$ so $T(1) = 0$ and $n \lg n = 0$.

**Inductive hypothesis.** $T(2^k) = k2^k$
Inductive step: Show that $T(2^{k+1}) = (k + 1)2^{k+1}$.

\[
T(2^{k+1}) = 2T(2^{k}) + 2^{k+1} \\
= 2 \cdot k2^{k} + 2^{k+1} \\
= k \cdot 2 \cdot 2^{k} + 2^{k+1} \\
= k \cdot 2^{k+1} + 2^{k+1} \\
= (k + 1) \cdot 2^{k+1}
\]

Next week I will pick up with this analysis and, no doubt, answer questions about the algorithm. I hope to start into chapter 5. See you then.